

**AFRL-IF-RS-TR-2003-145**  
**Final Technical Report**  
**June 2003**



# **CODE OPTIMIZATION FOR EMBEDDED SYSTEMS**

**Rice University**

**Sponsored by**  
**Defense Advanced Research Projects Agency**  
**DARPA Order No. F297, J468**

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

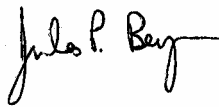
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

**AIR FORCE RESEARCH LABORATORY**  
**INFORMATION DIRECTORATE**  
**ROME RESEARCH SITE**  
**ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

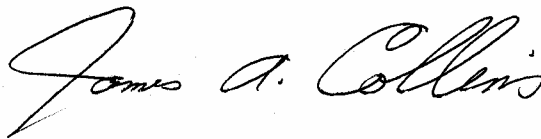
AFRL-IF-RS-TR-2003-145 has been reviewed and is approved for publication.

APPROVED:



JULES BERGMANN, Capt, USAF  
Project Engineer

FOR THE DIRECTOR:



JAMES A. COLLINS, Acting Chief  
Information Technology Division  
Information Directorate

<b>REPORT DOCUMENTATION PAGE</b>			<i>Form Approved</i> <i>OMB No. 074-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
<b>1. AGENCY USE ONLY (Leave blank)</b>		<b>2. REPORT DATE</b> Jun 03	<b>3. REPORT TYPE AND DATES COVERED</b> Final Jul 97 – Jul 01	
<b>4. TITLE AND SUBTITLE</b>  CODE OPTIMIZATION FOR EMBEDDED SYSTEMS			<b>5. FUNDING NUMBERS</b> C - F30602-97-2-0298 PE - 62301E PR - D002 TA - 02 WU - P6	
<b>6. AUTHOR(S)</b>  Keith D. Cooper, Devika Subramanian, Linda Torczon				
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Rice University Dept of Computer Science 6100 Main Street, MS 132 Houston, TX 77005			<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>  N/A	
<b>9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b>  Defense Advanced Research Projects Agency      AFRL/IFTC 3701 North Fairfax Drive                                      26 Electronic Pky Arlington, VA 22203-1714                                      Rome, NY 13441-4514			<b>10. SPONSORING / MONITORING AGENCY REPORT NUMBER</b>  AFRL-IF-RS-TR-2003-145	
<b>11. SUPPLEMENTARY NOTES</b>  AFRL Project Engineer: Jules Bergmann, IFTC, 315-330-2244, <a href="mailto:bergmannj@rl.af.mil">bergmannj@rl.af.mil</a>				
<b>12a. DISTRIBUTION / AVAILABILITY STATEMENT</b> Approved for public release; distribution unlimited.				<b>12b. DISTRIBUTION CODE</b>
<b>13. ABSTRACT (Maximum 200 Words)</b> This project investigated a number of problems that arise in compiling application code for embedded systems. These systems present the compiler with a number of challenges that arise from economic constraints, physical constraints, and idiosyncratic requirements of the application and processors. The project developed new techniques in optimization and code generation that addressed problems including code size reduction, instruction scheduling, data placement (on partitioned register set machines), spill code reduction, and operator strength reduction. It also produced fundamental work on transformation ordering.				
<b>14. SUBJECT TERMS</b> Application Code, Embedded Systems, Compiler-generated Code, Spill Code, Architectural Idiosyncrasies, Novel Optimization Paradigms				<b>15. NUMBER OF PAGES</b> 19
				<b>16. PRICE CODE</b>
<b>17. SECURITY CLASSIFICATION OF REPORT</b>  UNCLASSIFIED	<b>18. SECURITY CLASSIFICATION OF THIS PAGE</b>  UNCLASSIFIED	<b>19. SECURITY CLASSIFICATION OF ABSTRACT</b>  UNCLASSIFIED	<b>20. LIMITATION OF ABSTRACT</b>  UL	

## **Abstract**

This project investigated a number of problems that arise in compiling application code for embedded systems. These systems present the compiler with a number of challenges that arise from economic constraints, physical constraints, and idiosyncratic requirements of the application and processors. The project developed new techniques in optimization and code generation that addressed problems including code size reduction, instruction scheduling, data placement (on partitioned register set machines), spill code reduction, and operator strength reduction. It also produced fundamental work on transformation ordering.

## Table of Contents

Abstract	i
Table of Contents	ii
Summary	1
Introduction	2
Methodology	3
Results and Discussion	4
Summary and Conclusions	10
Annotated Bibliography	11

Additional material is available at <http://www.cs.rice.edu/~keith/Embed>, including papers, technical reports, and slides from various talks and presentations.

## 1. Summary

This project investigated a number of problems that arise in translating computer programs for execution on embedded computer systems—compiling those programs. Embedded systems are characterized by a number of constraints that do not arise in the commodity computer world. Most of these constraints have an economic basis. Embedded computers typically have limited amounts of memory. They often employ idiosyncratic processors that have been designed to maximize their performance for a limited class of applications. The applications are often quite sensitive to performance.

Some of the problems that arise in compiling code for execution on embedded systems have solutions that are relatively local in their impact within a compiler. For example, teaching the compiler to emit code for specialized instructions on a particular processor is easily handled during instruction selection — a modern code generator, based on pattern matching, can be extended to make good use of special case operations. We investigated several of these local problems. Other problems, however, have solutions that cut across the entire compiler. We tackled several of these cross-cutting problems. In both realms (local problems and cross-cutting problems), we developed an understanding of the issues involved, did some fundamental experimentation, proposed new techniques to address the problem, and validated those techniques experimentally.

To transfer the results of this work into commercial practice, we have published papers, distributed code, communicated with industrial compiler groups, and sent students to work in those groups. The techniques developed in this project are beginning to appear in the systems of other compiler groups — both research and commercial compilers. We expect more of them to be adopted in the future.

### *Major Results*

- ♦ New methods for reducing the size of compiler-generated code
- ♦ New techniques for instruction scheduling — both better schedulers for space constrained environments and stronger schedulers for hard problems
- ♦ A new algorithm for scheduling and data placement on processors with partitioned register sets — an increasingly popular feature in embedded processors
- ♦ New techniques for reducing the amount of spill code generated by a graph coloring register allocator and for reducing the impact of that spill code (both space and time)
- ♦ New techniques for some of the fundamental analyses and transformations used in code optimization for both embedded systems and commodity systems
- ♦ A new approach to building self-tuning optimizing compilers, which we call adaptive compilation.

## 2. Introduction

The embedded environment presents unusual challenges to a compiler. These systems are characterized by small memories, aggressive and idiosyncratic microprocessors, performance sensitive applications, and real-time applications. All too often, the available compilers fail to satisfy either the space or performance requirements, and the user must write at least part of the system in assembly code. While this works today, we will soon need better ways of building these systems. The rapid growth in the embedded systems marketplace, both applications and processors, suggests that not enough assembly-code wizards will be available to meet demand. Furthermore, within a hardware generation, the processors used in embedded systems will be complex enough to render effective assembly programming by humans virtually impossible.

Some of the problems that arise in targeting embedded systems have solutions that are relatively local in their impact within the compiler. For example, adding a specialized boolean instruction to the compiler's repertoire is an issue for instruction selection, easily handled by a technique like BURG. The more difficult problems have solutions that cut across the entire compiler. Our particular interest is in these cross-cutting problems: developing an understanding of the issues involved, proposing techniques to address the problems, validating the ideas experimentally, and working to move the solutions into commercial practice.

This project had three primary themes:

1. *Novel optimization paradigms* —

The resource, performance, and timing constraints of embedded systems suggest that more powerful compile-time techniques could be applied profitably during the final stages of program development if the compiler were allowed a constant factor more time. In this investigation, we looked at ideas that included: pursuing multiple optimization strategies and keeping the best result, using randomized algorithms and restart to explore large, complex solution spaces, and fundamentally rethinking the organization of our compilers.

2. *Resource constraints* —

The memory systems in embedded systems are almost always too small. Reducing the memory requirements of compiled code requires a concerted effort from parser to code generator. We investigated several schemes for reducing code space as a code optimization problem. We also looked at one technique for reducing data-space requirements (reducing the footprint of spill code).

3. *Architectural idiosyncrasies* —

The microprocessor architectures used in embedded systems evolve rapidly to improve their performance. We examined several specific issues, including partitioned register sets, predicated instructions, local (non-cache) memories, and branch-delay slots.

The sections that follow describe our major results.

### 3. Methodology

Our goal for this project was to improve compilation techniques in use for embedded systems. To achieve this requires more than simply inventing new techniques that address the problems. It requires careful experimental validation of both the costs and the benefits of new techniques. It requires detailed engineering of the techniques to ensure their implementability and their practicality. (The new methods must fit into the commercial compiler. No commercial group will rewrite their entire compiler to accommodate some academic result.) It requires a mechanism for transmitting the high-level concepts, the low-level engineering details, and the implementation insights to the commercial implementor in a concise and useful form. Finally, it requires an aggressive effort to ensure that commercial implementors are aware of the new work.

Because we understand the difficulty of moving new techniques into commercial practice, we have structured our experimental methodology to help us address each of these concerns.

1. ***Problem identification*** — To find new research problems, we read and profile the output of existing compilers, and we talk to commercial compiler groups (TI, Motorola, Intel, HP, Microsoft, and others).
2. ***Preliminary exploration*** — To understand the importance of a problem and its amenability to solution, we perform an initial round of experiments. This might involve hand simulation of a transformation or the construction of a prototype implementation (often, using inefficient algorithms). If the results are promising, we continue.
3. ***Algorithmic development*** — To refine our ideas, we build a serious prototype that runs in our research compiler. In the prototype, we work out the algorithmic and engineering details required for acceptable compile-time performance. We use the prototype to test effectiveness against a collection of representative codes. This is an iterative process, where testing reveals further opportunities for improvement.
4. ***Publication and distribution*** — To make the results of the work widely available, we publicize them on several levels. We publish papers in appropriate journals and conferences. We make the implementation accessible via the web. We visit with commercial compiler groups and discuss their problems and our solutions.

Historically, we have achieved reasonable success in moving ideas and techniques from our lab into commercial compilers from many companies.



## 4. Results and Discussion

This project, which ran from July 1997 through July 2001, investigated a number of issues in code optimization and code generation for embedded systems. This section summarizes the results of our major research thrusts. The final subsection describes a number of algorithms that we developed as a result of these inquiries that do not fit into any of the major research thrusts. The annotated bibliography provides a running commentary on the various publications and technical reports that we produced.

### *Novel Optimization Paradigms*

Historically, compilers operate by applying a fixed sequence of translation steps in a fixed order. This is true on the macro level; compilers generally run their optimizations in a fixed order. It is also true on a micro level; most individual transformations attack the opportunities for improvement in a deterministic order. The compiler confronts a problem: what is the best code to generate for the source program being translated? The compiler constructs an approximation to the best answer — the code is correct but not optimal. This approach is a sensible response to the constraints under which compilers have historically operated: produce correct code quickly.

As part of this project, we explored what might be possible if we relaxed these constraints. In particular, we relaxed the constraint that the compiler itself runs quickly. This created the option of using techniques that tried multiple approaches, evaluated the results, and kept the best code — an idea accepted in register allocation since the late 1980s. We applied this notion to two problems, with three sets of interesting results.

*Iterative Repair Scheduling* — Traditional instruction schedulers operate by using a greedy list-scheduling algorithm. While the folklore suggests that these schedulers do well in practice, there was little hard data that assessed how often list schedulers produce optimal schedules.

We built a series of schedulers based on an alternative paradigm, called iterative repair, and used these schedulers to understand the space of possible schedules and to measure the effectiveness of list scheduling. Iterative repair schedulers operate by constructing a gross approximation as an initial schedule. (The initial schedule must respect the data dependences, but not the resource constraints.) To transform the initial schedule into a valid schedule, the iterative repair framework chooses a mis-scheduled operation at random and places it in a position where it can legally execute. By restarting the algorithm multiple times, the framework can construct many distinct schedules. (This combination of randomization and restart is a powerful tool for exploring the space of schedules.) It can either gather data about the various schedules or it can simply keep the best schedule. Using different heuristics to select the next repair site produces distinct scheduling regimes.

Our experiments showed that:

1. List scheduling produces schedules of optimal length most of the time (more than ninety percent of the time).
2. A randomized version of list scheduling, run perhaps ten times, outperforms any single version that we tested.

3. Iterative repair can find schedules that consume fewer resources than those produced by list scheduling, even when list scheduling finds an optimal length schedule. For example, it often finds schedules that use fewer registers.
4. Blocks where list scheduling fails to find optimal results fall in a narrow range for one measurable parameter — available parallelism per issue slot. When this value falls within that range, it may be worth invoking an iterative repair scheduler. (The compiler can measure this parameter during list scheduling.)

*Computing Transformation Orders* — We conducted a series of experiments with transformation ordering. In the first, we built a simple genetic algorithm to find an ordering for the compiler’s transformations that produced compact programs. The genetic algorithm was able to reduce code size by an average of thirteen percent over the default optimization sequence in our compiler. (In contrast, direct compression using pattern matching and procedure abstraction produced an average of five and one-half percent in the same compiler. See references 2 and 3.)

Studying the strings that resulted from the genetic algorithm allowed us to derive a standard transformation sequence for compact code that achieved most of the benefit of running the genetic algorithm. It achieved an average of eleven percent reduction in code size when compared to the standard transformation sequence used in our research compiler. In the case of this problem (and, perhaps, these benchmarks), we were able to generalize from the experiment to discover a more broadly applicable sequence.

Based on our experience using genetic algorithms to compute transformation sequences for compact code, we expanded our inquiry to look at other objective functions, to explore more effective genetic algorithms, and to investigate other search techniques. This line of inquiry has produced an independent, NSF-funded research program (“Building Practical Compilers Based on Adaptive Search”, \$1.6 million, 8/2002 through 8/2007). That project will explore a number of issues, including better search techniques, the relationship between program properties and the “best” sequences, how to apply the results in a time-constrained compiler, and how to engineer compilers so that their passes can be reordered. (Reference 9 describes some of the early experiments on this project.)

### ***Dealing with Constrained Resources***

Resource constraints are a striking difference between the embedded environment and more general computing environments. The limited program and data memories found in embedded systems are driven by economics, as well as power and size constraints. These constraints are unlikely to ease in future.

We explored a number of techniques for reducing the memory requirements of compiled code. In general, two approaches make sense. The first is a direct attack — compressing the compiled code. The second is indirect — building compilers that generate smaller code in the first place. We worked on both problems. Finally, our work on architectural idiosyncrasies included techniques to reduce the data-memory footprint of programs.

*Compressing Compiled Code* — We built a code compressor based on ideas put forth in a 1984 paper by Fraser, Myers, and Wendt. The tool built a suffix-tree to represent the

program and identify disjoint but identical code sequences. It used this knowledge to shrink program size by creating a lightweight procedure to hold the common code and replacing each occurrence of the original sequence with a call to the lightweight procedure. This transformation is called “procedure abstraction.”

We extended the original work by abstracting register names and branch targets to increase the number of common sequences that it can identify. This produced an average five and one-half percent reduction in size on highly optimized code. (Optimized code is harder to compress because it has fewer common sequences. We can increase the percent compression by turning off optimizations; however, the final code will be slower and larger.) Because every reduced code sequence is longer (by the procedure call), the compressed code is slower than the original code. On average, the reduction was roughly one-half percent in speed for each one percent reduction in code size. Using profile information to avoid compressing frequently executed code can reduce the speed penalty. (See reference 2).

*Generating Smaller Code* — We looked at two different approaches to generating smaller code. As described earlier, we used genetic algorithms to find transformation sequences that produce compact programs. Additionally, we looked at scheduling techniques that avoid the code-growth problems associated with aggressive instruction scheduling.

Instruction scheduling reorders the operations in a program to hide both operational and memory latencies. If the scheduler restricts its attention to straight-line code, this process does not, in general, increase code size. Once the scheduler looks at multiple blocks (a process called non-local scheduling), code growth becomes an issue. If the scheduler moves an operation across a branch, it may need to insert code elsewhere to compensate for the branch’s new location. (Moving an operation into an earlier block may force it to execute on paths where it previously did not. This may require compensation code. Moving an operation into a later block may remove it from some paths. This may require compensation code.)

We developed a pair of algorithms that perform non-local instruction scheduling without requiring any compensation code. These algorithms constrain the scheduler to only move operations across branches when they do not require compensation code. These algorithms produce an average twelve percent improvement in running time with no growth in code size. (See reference 1.)

*Combining the Two Approaches* — The compiler could combine these two approaches and apply compression techniques to the smaller code. A more interesting experiment would be to fix the code compressor on the end of a transformation sequence and use the genetic algorithm (or another biased search technique) to find sequences that produce the smallest code for individual applications. This approach would use the adaptive approach to generate code that compresses well. Breaking ties by estimated or measured running time should mitigate any negative effects on execution time.

*Coloring Spill Memory* — As part of an investigation into effective uses for the local, non-cache memories found on DSPs such as the Texas Instruments C6000 series, we developed techniques for coloring the storage locations used to hold spilled values. This work showed that we could significantly reduce the amount of data memory needed to hold spilled values. The same techniques might be applied to reduce space requirements

for global and static values; however, this would require whole program analysis of value lifetimes — an analysis performed by few systems.

### ***Architectural Idiosyncracies***

Many, if not most, embedded systems use specialized processors that have been optimized for lower power consumption, for specific applications, or for specific environments. These processors often include features that are not found in commodity microprocessors, such as local memories with disjoint address spaces (as opposed to classic caches), functional units with specialized operations, and long pipelines (like the five-cycle branch latency on TI's C6000.) In this research thrust, we looked at some of the problems that arise in embedded processors.

*Placement on Partitioned Register-set Machines* — As the number of functional units in a processor increases, the number of ports into its register set must also grow. An ALU, for example, typically reads two registers and writes one during each operation. At some point, the logic required to multiplex all these values back and forth between functional units and registers becomes so complex that it limits the processor's base cycle time. To address this issue, architects can use multiple, disjoint register sets.

The resulting machine typically has two to four functional units per register set. We call this combination of functional units and their register set a *partition*. These machines usually have a limited ability to move data directly between register sets — either a direct register-to-register copy operation or the ability for some operations to reference a register in another register set. The bandwidth for these inter-partition references is usually one off-partition register per cycle.

Partitioning the register set reduces the number of ports per register set. However, it complicates the programming model. The code generator for a partitioned register set machine must consider both data placement and operation placement. Each operation must execute in a partition where its operands are available. Inevitably, some inter-partition movement must be inserted.

This problem is intricately related to both instruction scheduling and register allocation. As part of this project, we developed algorithms that combine instruction and data placement with scheduling. We performed experiments showing that our approach produced results as good or better than the prior art — Ozer's algorithm. We used the experimental framework to assess the benefit of giving each partition one or more registers dedicated to holding the results of inter-partition copies. (See reference 8.)

*Exposed pipelines*— A major emphasis of this entire project was understanding and improving techniques for instruction scheduling. In the section on constrained resources (above), we described two new techniques for non-local scheduling: extended basic block scheduling and dominator-path scheduling. In the section on novel optimization paradigms (above), we described our work that used the iterative-repair paradigm to build a family of powerful (albeit slow) schedulers. Our work with partitioned register set machines used local list scheduling, in both forward and backward form.

While these approaches might appear to be disconnected, they can, in fact, be combined. The resulting scheduler would operate on either extended basic blocks or dominator paths. It would incorporate the constraints on inter-block motion that eliminate the need

for compensation code. It would use randomization and restart to schedule the region multiple times. Finally, it would apply both forward scheduling and backward scheduling. Our results suggest keeping the best result from five forward and five backward randomized passes over the region.

*Predication* — One of our initial goals was to study the impact of predication on both the shape of code that the compiler should produce and on the optimizations that the compiler writer should include in the compiler. While we devoted some effort to studying this problem, we made no significant or novel discoveries in this area.

*Combining allocation and scheduling* — Compiler writers have long recognized that instruction scheduling and register allocation interact. Unfortunately, both problems are NP-complete on their own. Attempts to combine them have generally involved underallocating to provide the scheduler with extra leeway in moving operations, or in limiting code motion so that it does not increase register pressure.

(One interesting approach, from a 1998 paper by Goodman and Hsu, uses two schedulers: an aggressive scheduler that ignores register constraints and a conservative scheduler that tries to reduce demand for registers. The Lee and Hwu approach schedules aggressively until demand for registers matches the number of available registers, then switches to the pressure-sensitive scheduler until enough free registers are available for the aggressive scheduler to continue.)

Our work with iterative-repair schedulers led us to a novel approach for integrating scheduling with allocation. We took our iterative repair schedulers and added a “repair move” that reduces register pressure. This framework recognizes two kinds of faults in a schedule — too many operations in a single cycle and too many live values in a given cycle. It can cure the former by picking an instruction from that cycle and scheduling it at another point in the code. (Data dependences determine whether or not it can be moved earlier. It can always be moved later.) It can cure the former by picking an instruction from that cycle and scheduling it at a point in the code where it reduces demand for registers. (Moving a last use of some value closer to its definition shrinks its lifetime. Moving a definition closer to its first use shrinks its lifetime.)

This framework may be the first combined scheduler and allocator that makes an unbiased tradeoff between the conflicting demands of these two hard problems. The results, reported in Schielke’s thesis (reference 10) are quite encouraging. It found opportunities in twelve percent of basic blocks and thirty-three percent of extended basic blocks. While the average-case speed improvement was three to five percent, the best case was in excess of forty-one percent.

### ***Improvements in Basic Analysis and Optimization***

As part of our investigation, we developed a number of improvements to classic analysis and optimization techniques. While these were not part of our primary research thrusts, they address important problems that have application in both the embedded systems arena and in the commodity systems arena.

*Operator Strength Reduction* — We developed, with partial support from this project, a simple and elegant algorithm for performing operator strength reduction. It uses properties of the static-single assignment form (SSA) of the program to create an

algorithm that is easy to understand and simple to program. It produces, in a single pass, results that are equivalent to multiple applications of the classic Allen-Cocke-Kennedy algorithm (the best prior art). This algorithm has been implemented in several commercial systems. (See reference 11.)

*Dominance Calculations*—A fundamental step in building SSA form for a program involves computing dominators and dominance frontiers. We developed a simple algorithm for the dominance calculation that is, in practice, the fastest known method for this problem. Other algorithms for this problem have lower asymptotic complexity. However, careful engineering of the data structures for our algorithm let us move the cross-over point (where the asymptotically faster algorithms begin to win) out beyond graphs of thirty-thousand nodes. Since most control-flow graphs are orders of magnitude smaller than this, our algorithm outperforms the asymptotically better ones in practice. This work is described in a paper currently in review (see reference 12); the algorithms and data structures are described in Cooper and Torczon's forthcoming book.

*Building Control-flow Graphs* — In our work on partitioned register set machines, we became interested in object-code to object-code translation for Texas Instruments' C6000 series of processors. Unfortunately, those processors allow branches to be scheduled in the delay slots of other branches. Since they have five delay cycles per branch, the TI compiler for the C6000 aggressively uses this feature when it software pipelines small loops. (It stuffs the pipeline with branches, then places the single execute packet in the last delay slot.)

These programs break all previous algorithms for building control-flow graphs. For example, the compiler cannot determine where basic blocks begin and end without performing an iterative computation to discover which branches might be pending on entry to the block. We developed an algorithm that builds correct control-flow graphs for assembly code on machines with this feature. This is a necessary first step in building object-level tools for architectures like the C6000. (See reference 13.)

*Reducing the Impact of Spill Code* — Register allocation continues to be an active research issue. We developed several new strategies for reducing the amount of spill code generated by a graph-coloring register allocator. Some of them depend on specific features of embedded processors. Others have more general application. We continue to work and to publish actively in this area.

## 5. Summary and Conclusion

This project developed a number of novel techniques for improving the code that compilers can generate for embedded systems.

- ♦ We worked on several aspects of instruction scheduling. This work should find practical application in compilers for embedded systems and for conventional systems based on commodity microprocessors.
- ♦ We developed several new approaches to reducing code size, and did some experiments that used coloring to reduce the program's footprint in data memory.
- ♦ We did fundamental new work that combined techniques from other areas, such as randomization and restart, genetic algorithms, and iterative repair, to important problems in optimization and code generation.
- ♦ We developed new techniques, like OSR, that have broad application. For example, the CFG reconstruction work is a first step toward object-level optimization on DSP instruction sets — optimization for properties such as power reduction or code size reduction.

The work on novel optimization paradigms will take longer before it has an impact on commercial practice, although we have seen some groups using techniques like our genetic algorithms to choose compiler option flags in program-specific ways. We have received additional funding to continue our investigation of these ideas.

## 6. Annotated Bibliography

### *Work on Code Space (Memory constraints)*

1. "Non-local instruction scheduling with limited code growth." Keith D. Cooper and Philip J. Schielke. 1998 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), Montreal, CA, June 1998. (Lecture Notes in Computer Science, 1474, F. Mueller and A. Bestavros, (editors), Springer), June 1988, pp. 193-207.

*This paper presented a scheduling algorithm that produced roughly twelve percent faster execution with zero code growth. Prior art for scheduling either confined itself to straight-line code (local scheduling) or inserted compensation code between blocks (producing code growth). This raised the bar for schedulers, even in a code-space constrained environment.*

2. "Enhanced Code Compression for Embedded RISC Processors." Keith D. Cooper and Nathaniel McIntosh, Proceedings of the SIGPLAN 99 Conference on Programming Language Design and Implementation (PLDI), SIGPLAN Notices 34(5), pages 139-149.

*This paper revisits and improves a code compression algorithm published by Fraser, Myers, and Wendt in the 1984 SIGPLAN Compiler Construction Conference. The algorithm uses a suffix tree to identify common code sequences and replaces multiple occurrences of a common sequence with an inexpensive call to a single copy. The extensions include abstracting register names and branch targets to allow detection of more common sequences, and an algebraic approach to handling overlapping matches.*

*Bottom line: an average of five and one-half percent compression on highly optimized code. [Much larger results are possible if we turn off prior optimization.] The improvement costs about one-half percent slowdown for each one percent reduction in code space. Profiling can reduce the cost (by avoiding compression of hot code sequences). However, this reduced the amount of compression achieved.*

3. "Optimizing for Reduced Code Space using Genetic Algorithms." Keith D. Cooper, Philip J. Schielke, and Devika Subramanian, 1999 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), Atlanta, GA, May 1999.

*This paper shows an alternative approach to code space problems: generate small code in the first place. The authors show (in the same compiler as the Cooper/McIntosh paper above) that choosing a good sequence for the compiler's optimizations can reduce code size by an average of thirteen percent — far more than we achieved with the suffix-tree approach. We built a simple genetic algorithm to discover optimization orders that produce compact code. It converged in several hours to produce small, fast programs. We used code size as our objective function, with program execution time as a tie-breaker. Thus, the compact programs were also, in general, faster than the programs produced by the compiler's default sequence (which aimed for fast code rather than compact code).*



*Studying the results of our experiments led us to derive a single fixed optimization order which produced most of the benefits (roughly eleven percent reduction from the default compiler) while avoiding the cost of running the genetic algorithm. This paper has led to much further work in our own group (such as a recently funded NSF ITR project) as well as work by others.*

#### *Work on Architectural Idiosyncracies*

4. "Compiler-controlled Memory." Keith D. Cooper and Timothy J. Harvey. Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, CA, October 1998, pp. 2-11.

*This technique uses local memory, such as that found on the TI C6000 series machines, for spilling. It shows that a small amount of memory will suffice to hold spilled values and that using local memory can drastically reduce the traffic to main memory. Along the way, it shows how to color spill memory as a tool for reducing the footprint of spilled values.*

5. "An Experimental Evaluation of List Scheduling", Keith D. Cooper, Philip J. Schielke, and Devika Subramanian, Rice University, Department of Computer Science Technical Report 98-326, September 1998.

*See the description of Schielke's thesis (10 below).*

6. "Issues in Instruction Scheduling", Philip J. Schielke Rice University, Department of Computer Science Technical Report 98-323, September 1998.

*See the description of Schielke's thesis (10 below).*

7. "Reducing the Impact of Spill Code", Timothy J. Harvey, Masters Thesis, Rice University, Department of Computer Science, May 1998.

*This thesis describes three techniques for reducing spill code in a compiler that uses a graph-coloring register allocator. The first is a fix to one problem in Briggs' formulation of the problem — the NeedLoad calculation. The second is a postpass technique, called register scavenging, that finds segments of the code where registers are underutilized and promotes previously spilled values into those unused registers. The third technique uses local memory, such as that found on the TI C6000 series machines, for spilling. It shows that a small amount of memory will suffice to hold spilled values and that using local memory can drastically reduce the traffic to main memory. Along the way, it shows how to color spill memory as a tool for reducing the footprint of spilled values.*

8. "Cluster assignment and scheduling for partitioned register set machines," Jingsong He, Masters Thesis, Rice University, Department of Computer Science, April 2000.

*This thesis developed an algorithm for instruction scheduling combined with cluster assignment on partitioned register set machines. The algorithm improves on Ozer's original algorithm. The thesis looks at forward and backward scheduling. It also includes experiments that demonstrate the effectiveness of dedicating one or more registers per partition to intercluster copy operations.*

### *Work on Novel Optimization Paradigms*

9. "Adaptive Compilers for the 21st Century", Keith D. Cooper, Devika Subramanian, and Linda Torczon, Proceedings of the 2001 Los Alamos Computer Science Institute Symposium, Santa Fe, NM, USA, October 16–17, 2001.

*This paper, which also appeared in the Journal of Supercomputing, outlines a framework for building adaptive compilers — compilers that reconfigure themselves in response to the input program and a user-selected objective function. It also provides a preliminary report on a massive experiment intended to characterize (for the first time) the space in which such an adaptive compiler operates — the multidimensional space of interactions between specific optimizing transformations.*

*This work grew out of the code compression work described in “Optimizing for Reduced Code Space Using Genetic Algorithms” It will continue in our NSF funded ITR project (\$1.6 million for five years, starting 8/2002).*

10. "Stochastic Instruction Scheduling", Philip J. Schielke, Ph.D. Thesis, Rice University, Department of Computer Science, (available as TR 00-370), November 2000

*This thesis examines instruction scheduling. The dominant algorithm for this problem, greedy list scheduling, is a constructive algorithm that builds and returns an approximate solution. In this work, we built a series of schedulers based on the iterative repair paradigm and used them to assess 1.) how effective list scheduling actually is and 2.) how much room remains for improvement.*

*As a side issue, the iterative repair framework also creates the first fair framework for trading off register allocation against scheduling. Schielke’s experiments provide results that suggest this approach should be pursued.*

*Bottom line: List scheduling does well in practice, finding an optimal schedule most of the time. Iterative repair can find equivalent schedules that use fewer registers. When list scheduling misses the optimal schedule, iterative repair often finds it. The thesis describes a measurable property that predicts when list scheduling is likely to have problems. This might be used as a heuristic to let the compiler decide when it is appropriate to invoke the stronger, more expensive scheduler.*

(See also "Optimizing for Reduced Code Space using Genetic Algorithms" and "An Experimental Evaluation of List Scheduling" )

### *Background Work on Compilation Issues*

11. "Operator Strength Reduction.” Keith D. Cooper, L. Taylor Simpson and Christopher A. Vick. ACM Transactions on Programming Languages and Systems, 23(5), September, 2001.

*This paper presents a simple, elegant algorithm for Operator Strength Reduction. It relies on detailed properties of the static single assignment form to create a fast, effective algorithm. The algorithm produces the same results as the classic Allen Cocke Kennedy algorithm (the strongest prior algorithm) without the need to iterate the entire algorithm for second order effects. The algorithm is easy to implement, easy to teach, and easy to understand.*

12. "A Simple, Fast Dominator Algorithm", Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. (will eventually appear in Software—Practice and Experience) (The basic results from the paper influenced the algorithms presented for this problem in Kennedy's 2001 Book from Morgan-Kaufmann and in Cooper and Torczon's book (to appear, Morgan Kaufmann, 2003).

*This paper shows how careful engineering of a simple, easily understood algorithm can produce a method for solving the dominator equations that runs more quickly than the algorithm with the best asymptotic complexity. The importance of the paper lies in the critical role that dominance information plays in advanced program transformations (for example, static single assignment form builds on dominance). This paper should influence the way that we build compilers, and it should lower the entry cost, in implementation effort and intellectual effort, for building high-quality optimizing compilers.*

13. "Building a Control-flow Graph in the Presence of Delay Slots", K.D. Cooper, T.J. Harvey, and T. Waterman, Computer Science Department Technical Report.

*The fundamental data structure that underlies all program analysis is the control-flow graph (CFG). In systems that attempt to analyze and improve already-compiled code, the CFG must be inferred from the object-level form of the program. If the underlying architecture supports delay slots on branches, the task is more complex than the traditional compiler-based algorithm can handle. If those delay slots can contain branches, as on the Texas Instruments C6000 series of processors, then the problem becomes much harder. This paper presents an algorithm for building the CFG in an object-level analyzer for the C6000.*